Study of reverse engineering protections on Android applications

Master in Cybersecurity Management (2019-2020)

Jaume Serrats

Objectives and environment

Reverse engineering protections are code checks that the developer inserts into the application in order to try to make it more difficult for an attacker (the reverse engineer) to understand how the application works, or to modify its logic. For example a banking application has to protect its business logic, their clients information and its own integrity to avoid an attacker modifying the application and distributing a malicious version. Note that malicious software usually also uses reverse engineering protections for the same reasons, since if the functioning of the application is unknown it is harder to detect and protect the users from them.

The main objective in this project will be to investigate about reverse engineering protections design and evasion on an Android environment. Since this is a very broad field, I will be investigating a limited number of possible protections, chosen with a popularity and usefulness criteria. It is also important to understand that the reverse engineering field is a constant battle between the developer and the reverse engineer, so it is not possible to describe all the new technologies and evasions developed constantly. Because of this, I will try to describe the basics in order to get a general knowledge of each protection.

Work methodology

For each of the chosen protections, i have:

- Found or developed an open source implementation of the control in order to fully understand how it works by developing it or understanding the source code
- Investigated a way to bypass said protection.
- Investigate on future improvements or alternative protections that could mitigate or prevent the bypass.

This scenario is not fully realistic, since an attacker in a real world scenario would not know the implementation of the control, making it more difficult. Nonetheless, since this is an educational project I consider this to be a good start.

Tools

The tools used are pretty standard in the Android reversing community.

Frida Dynamic instrumentation toolkit

Frida is a very popular and powerful tool that allows us to hook into processes, modifying its flow and logic on the fly without the need to modify low-level code, or restart the application. This is very powerful when evading reverse engineering controls, Since they are often obfuscated and difficult to analyze statically.

Runtime Mobile Security (RMS)

RMS is a Frida web interface that makes workflow much quick and easier.

Android emulator from Android SDK

The emulator allows us to test all the controls in a realistic scenario. The emulator was chosen over a real phone because some experiments require having a rooted phone, which is generally not safe to have on a personal phone that is used daily.

Burp Proxy

Burp is a proxy focused on attacking web communications, edit and repeating requests, decoding data, and more.

Android Debugger Bridge (ADB)

ADB is a versatile command-line tool that lets you communicate with a device. The adb command facilitates a variety of device actions, such as installing and debugging apps, and it provides access to a Unix shell that you can use to run a variety of commands on a device

Apktool

Apktool is a tool for reverse engineering 3rd party, closed, binary Android apps. It can decode resources to nearly original form and rebuild them after making some modifications. It also makes working with an app easier because of the project like file structure and automation of some repetitive tasks like building apk, etc.

Work accomplished

I've chosen the following controls to study:

- root check
- emulator check
- certificate pinning
- binary obfuscation

These are four of the most usual controls that we can find implemented on applications that deal with sensitive information.

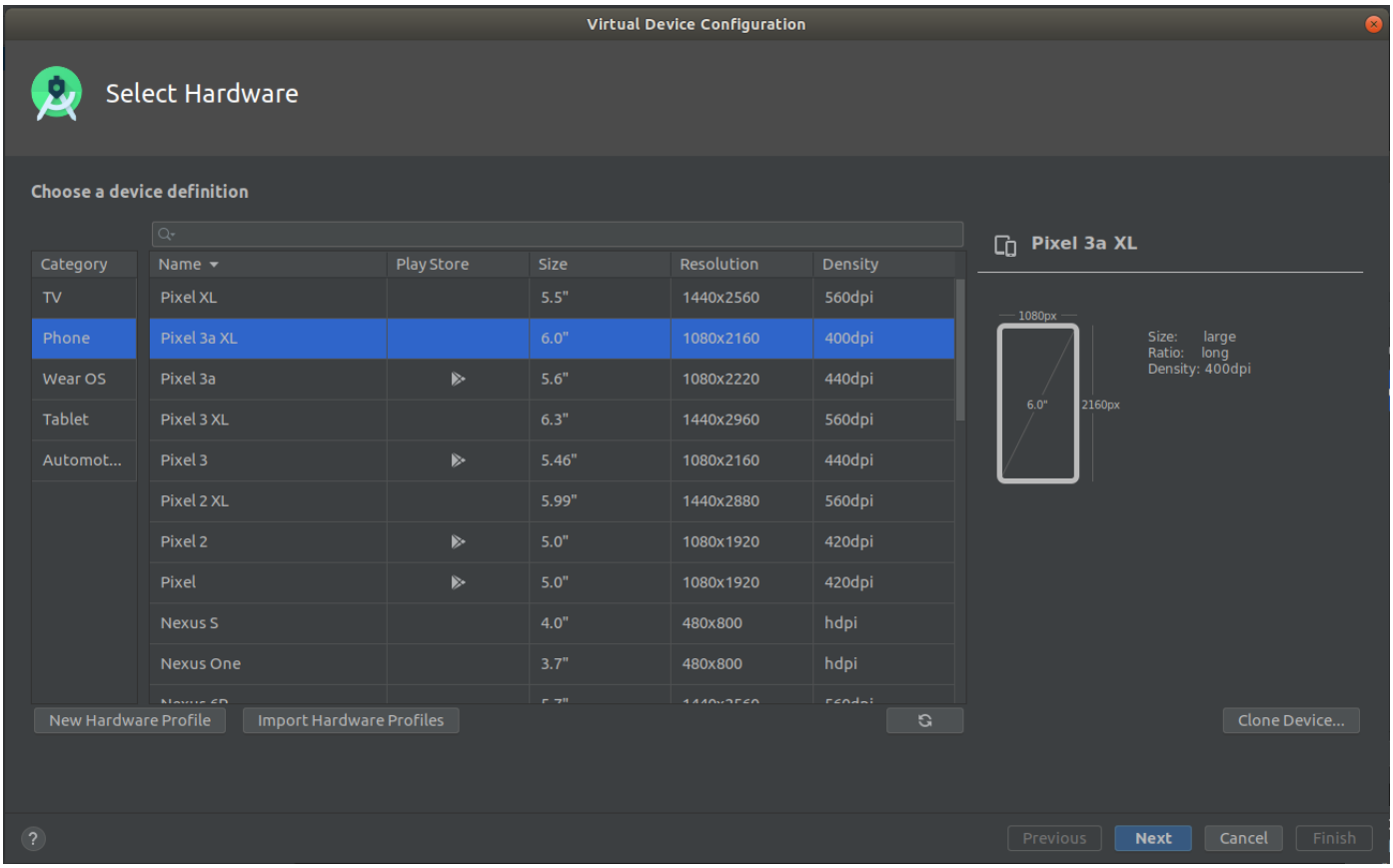
Root

The root account in a linux environment is the administrator of the device, and has all the permissions. Commercial Android smartphones usually don't allow access easily to this account, since it can be a security risk for the user, as a malicious application could escalate privileges, break the application sandbox and obtain full control of all the information stored and control all the processes running in the phone.

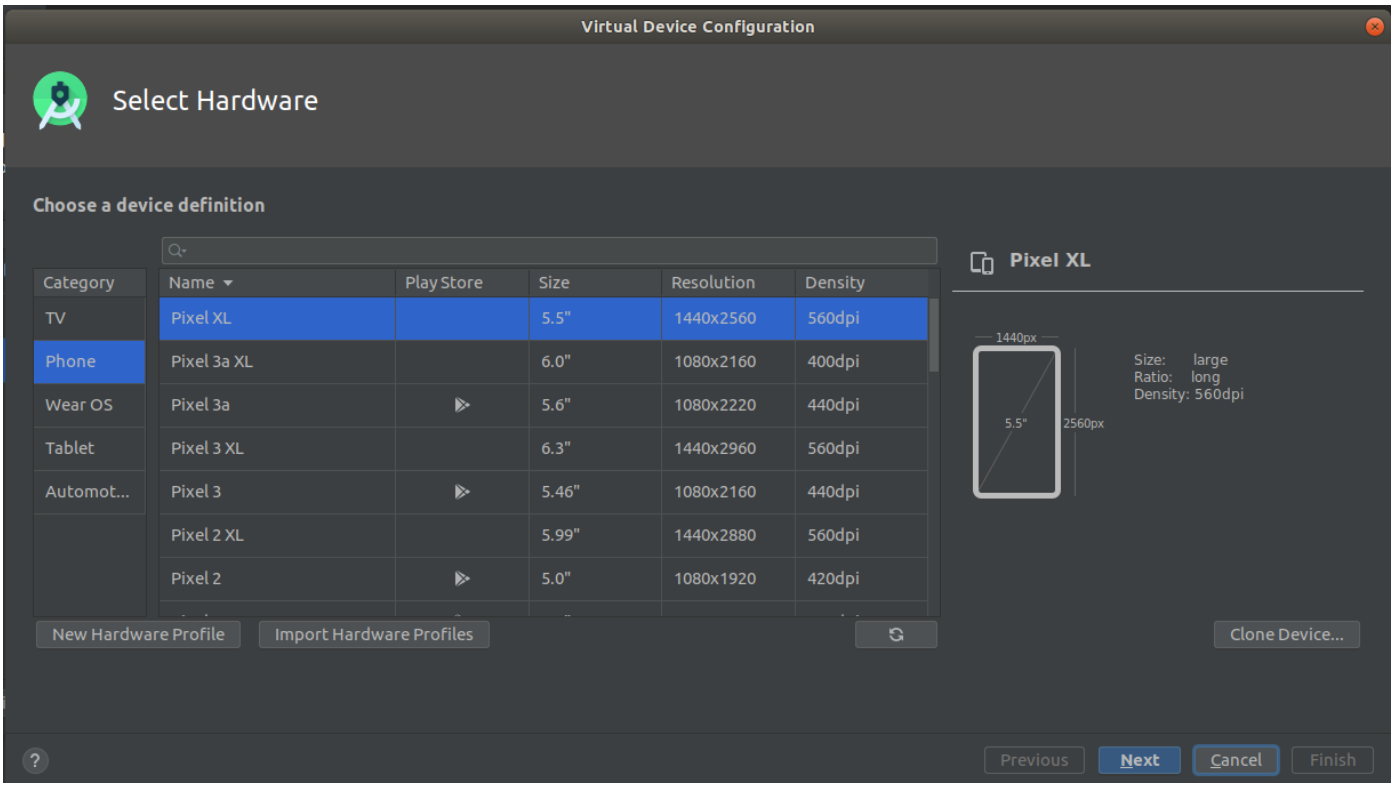
There are several ways to detect root in an Android device. Most of them are implemented in https://github.com/scottyab/rootbeer, an open source library mentioned in OWASP's MSTG. We'll base our evasion techniques on the controls implemented in this library.

Root environment

First we need a root environment. We'll use the default Android emulator



Choose a device without Google Play Services installed.



To obtain a root shell we first must type:

```
$ adb root
restarting adbd as root
```

and then launch the shell

```
$ adb shell
generic_x86_64:/ # whoami
root
```

We now have a device with full permissions. To launch the emulator from command line type:

```
~/Android/Sdk/emulator/emulator -avd <name of the avd created before>
```

Installation of RootBeer

Clone the git project

```
$ git clone git@github.com:scottyab/rootbeer.git
Cloning into 'rootbeer'...
remote: Enumerating objects: 33, done.
remote: Counting objects: 100% (33/33), done.
remote: Compressing objects: 100% (24/24), done.
remote: Total 1779 (delta 2), reused 20 (delta 0), pack-reused 1746
Receiving objects: 100% (1779/1779), 2.70 MiB | 2.79 MiB/s, done.
Resolving deltas: 100% (735/735), done.
```

Get into the folder and run `gradlew`. Here we are installing a debug version for simplicity, but in this case it does not matter. The debug version means that the APK file is signed using a debug key.

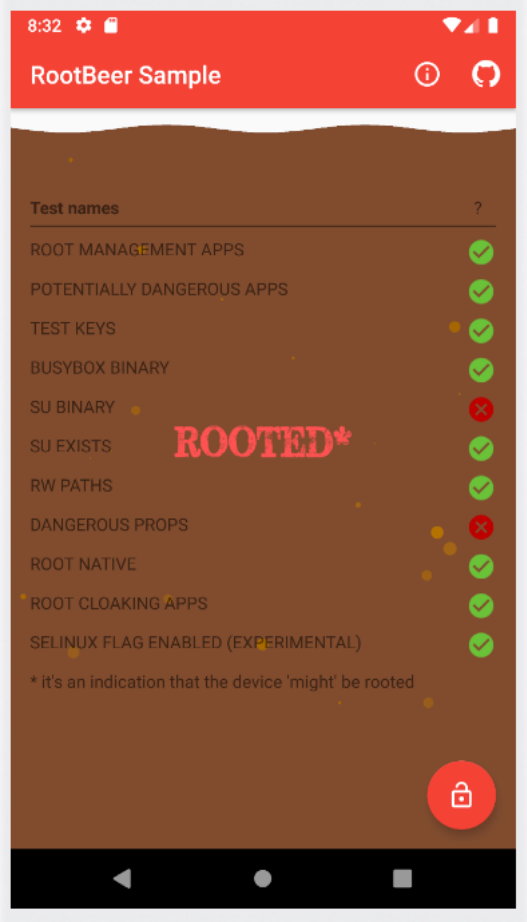
```
$ ./gradlew installDebug

> Configure project :rootbeerlib
Compatible side by side NDK version was not found.
```

```
> Task :app:stripDebugDebugSymbols UP-TO-DATE
Compatible side by side NDK version was not found.

> Task :app:installDebug
Installed on 1 device.

BUILD SUCCESSFUL in 6s
44 actionable tasks: 1 executed, 43 up-to-date
```

After this we'll have our app installed in the android drawer.

Base test

We'll first launch the RootBeer demo app on our root environment to see which controls detects.



We can see that our enviornment got detected because we have a su binary and modified properties file.

Bypassing binary detection

The su binary is an executable file that allows a normal user to escalate privileges into a root account. For example:

```
generic_x86_64:/ $ whoami
shell
generic_x86_64:/ $ su
generic_x86_64:/ # whoami
root
```

The presence of this binary is an obvious indicator that the system we are on is rooted.

In order to evade the detections we must first understand how we got detected. In the case of RootBeer, the library is open source, so we can directly look up the code.

Const.java

```
public static final String BINARY_SU = "su";
```

```
// ...

// These must end with a /
public static final String[] suPaths ={
        "/data/local/",
        "/data/local/bin/",
        "/data/local/xbin/",
        "/sbin/",
        "/su/bin/",
        "/system/bin/",
        "/system/bin/.ext/",
        "/system/bin/failsafe/",
        "/system/sd/xbin/",
        "/system/usr/we-need-root/",
        "/system/xbin/",
        "/cache/",
        "/data/",
        "/dev/"
};
```

RootBeer.java

```
public boolean checkForSuBinary(){
    return checkForBinary(BINARY_SU);
}

//...

public boolean checkForBinary(String filename) {

String[] pathsArray = Const.getPaths();

boolean result = false;

for (String path : pathsArray) {
    String completePath = path + filename;
    File f = new File(path, filename);
    boolean fileExists = f.exists();
    if (fileExists) {
        QLog.v(completePath + " binary detected!");
        result = true;
    }
}

return result;
}
```

As we can see the software looks for a file called `su` in the usual paths where it can be found in a normal installation. To see where we have the binary we can do the following:

```
generic_x86_64:/ # which su
/system/xbin/su
```

The `/system` path is mounted as read only, so we cannot modify it just yet. In order to do so in an emulator, we must start it with the flag `-writable-system`

```
~/Android/Sdk/emulator/emulator -avd root -writable-system
```

And then remount the partitions with

```
adb remount
```

It is important to maintain the `su` permissions and ownerships in order not to break anything

```
generic_x86_64:/system/xbin # ls -la su
-rwsr-x--- 1 root shell 11056 2019-04-11 00:54 su
```

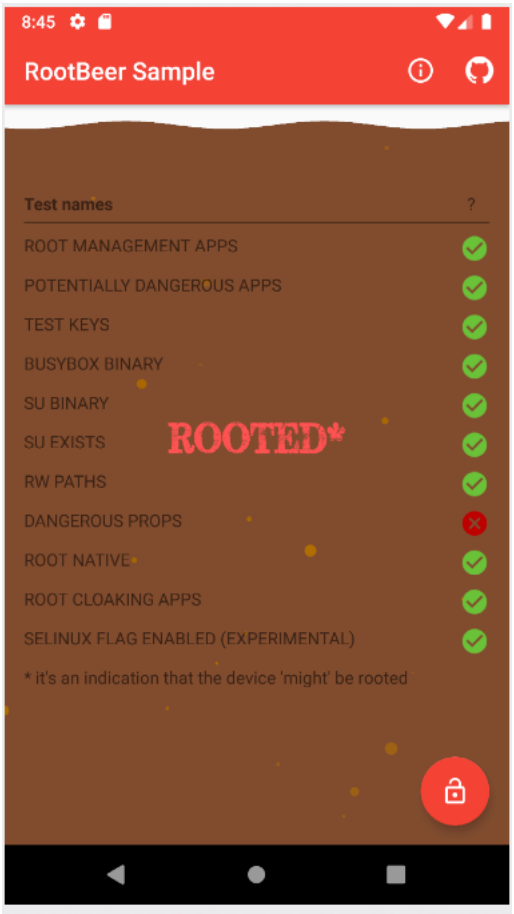
Now we can rename the su binary in order to avoid detection.

```
generic_x86_64:/system/xbin # mv su suu
```

Now we can call the binary su with its new name in order to become root.

```
generic_x86_64:/ $ suu
generic_x86_64:/ #
```

Meanwhile, the control is correctly bypassed



Bypassing dangerous properties detection

RootBeer control looks for 2 specific properties and returns True if any of them are set.

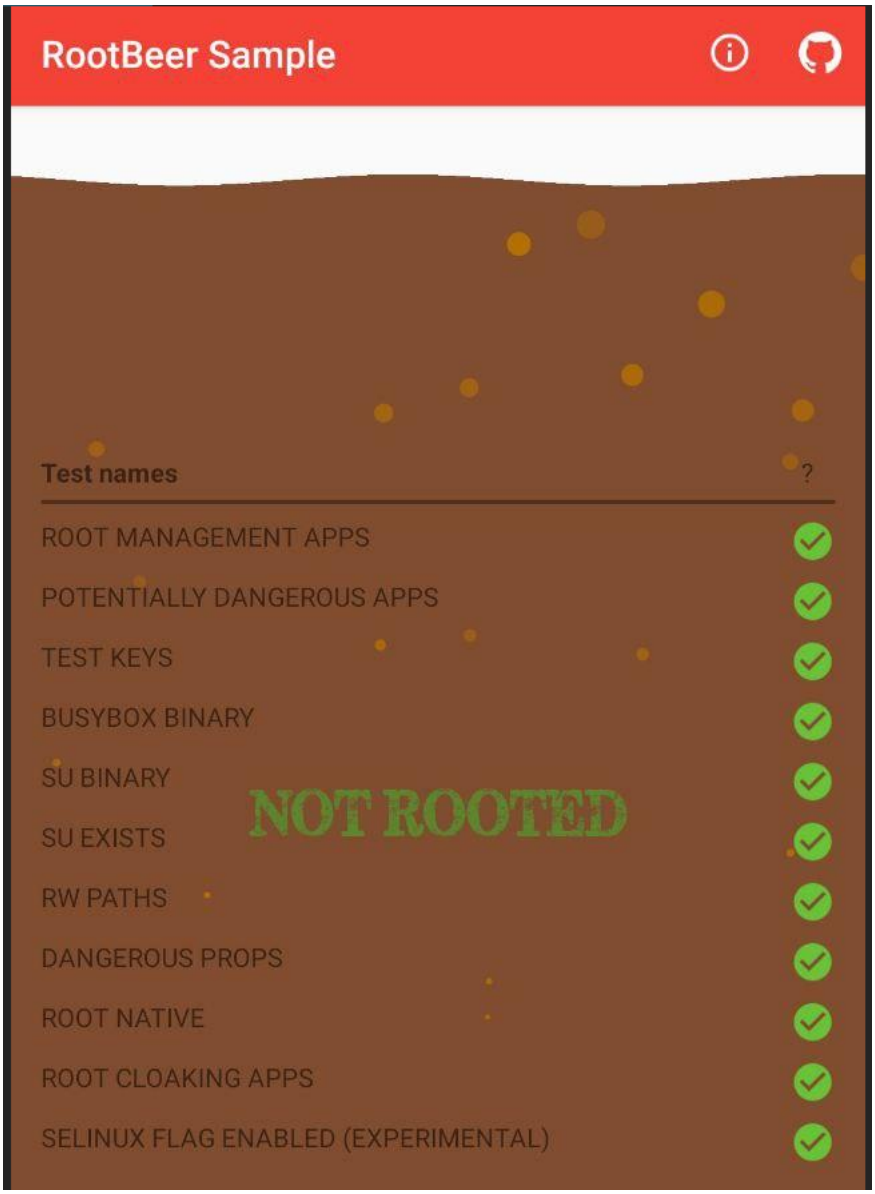
```
public boolean checkForDangerousProps() {

    final Map<String, String> dangerousProps = new HashMap<>();
    dangerousProps.put("ro.debuggable", "1");
    dangerousProps.put("ro.secure", "0");

    boolean result = false;

    String[] lines = propsReader();

    if (lines == null){
        // Could not read, assume false;
        return false;
    }

    for (String line : lines) {
        for (String key : dangerousProps.keySet()) {
            if (line.contains(key)) {
                String badValue = dangerousProps.get(key);
                badValue = "[" + badValue + "]";
```

```
                if (line.contains(badValue)) {
                    QLog.v(key + " = " + badValue + " detected!");
                    result = true;
                }
            }
        }
    }
    return result;
}
```

In oder to avoid the detection by the properties file, the easiest way is to hook the function with Frida. The following Frida script overloads the implementation of the methods `checkForSuBinary` and `propsReader` so both always return `false`. It also prints some useful debug information.

```
Java.perform(function () {
    var classname = "com.scottyab.rootbeer.RootBeer";
    var classmethod = "checkForSuBinary";
    var hookclass = Java.use(classname);

    //public boolean checkForSuBinary()

    hookclass.checkForSuBinary.overload().implementation = function () {
        send("CALLED: " + classname + "." + classmethod + "()");
        var ret = false;

        var s="";
        s=s+("\nHOOK: " + classname + "." + classmethod + "()");
        s=s+"\nIN: "+"";
        s=s+"\nOUT: "+ret;
        send(s);

        return ret;
    };
});

Java.perform(function () {
    var classname = "com.scottyab.rootbeer.RootBeer";
    var classmethod = "propsReader";
    var hookclass = Java.use(classname);

    //private java.lang.String[] propsReader()

    hookclass.propsReader.overload().implementation = function () {
        send("CALLED: " + classname + "." + classmethod + "()");
        var ret = false;

        var s="";
        s=s+("\nHOOK: " + classname + "." + classmethod + "()");
        s=s+"\nIN: "+"";
        s=s+"\nOUT: "+ret;
        send(s);

        return ret;
    };
});
```

As we can see, when hooking Frida to Rootbeer with the script, all checks return negative, and the app is not able to detect root.


RootBeer Sample
Test names
?
ROOT MANAGEMENT APPS
POTENTIALLY DANGEROUS APPS
TEST KEYS
BUSYBOX BINARY
SU BINARY
SU EXISTS
RW PATHS
DANGEROUS PROPS
ROOT NATIVE
ROOT CLOAKING APPS
SELINUX FLAG ENABLED (EXPERIMENTAL)
NOT ROOTED

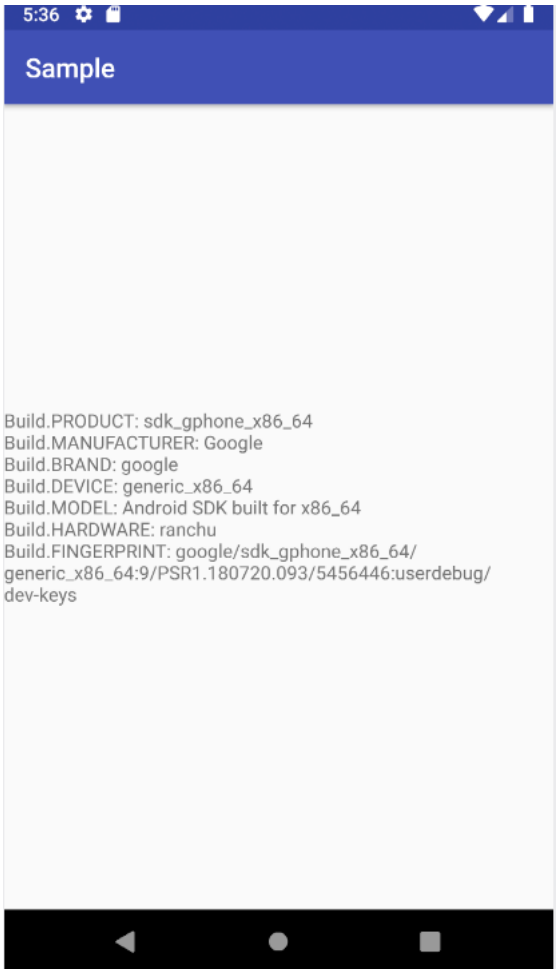
Emulator detection

In the context of anti-reversing, the goal of emulator detection is to increase the difficulty of running the app on an emulated device, which causes problems to some tools and techniques reverse engineers like to use. This increased difficulty forces the reverse engineer to defeat the emulator checks or utilize the physical device, making the analysis harder.

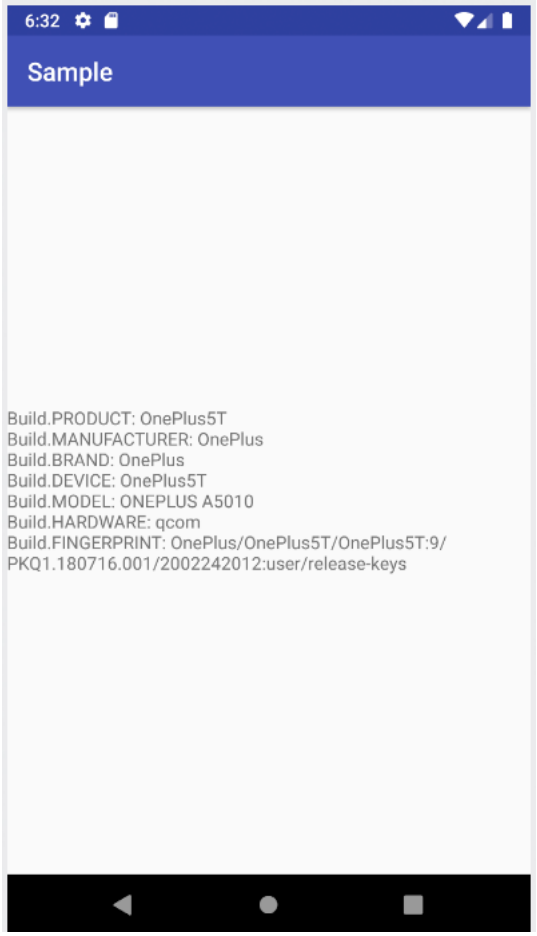
To check this, I have developed a fork from the Open source emulator checking app called Emulator Detector. The developed fork can be found here. This application basically checks for several properties in the file `build.prop` that are common on an emulator. This properties indicate parameters such as the name of the model, the manufacturer, and other similar information.

The original Emulator Detector application only returned a boolean result for the application runs in an emulated environment or not. The developed fork also prints in the screen the results for each parameter. This way we can easily see the parameters we are checking and its values.

The following screenshot shows the results of the app installed on our emulator without any tampering. As we can see, it is really obvious that the device is emulated.



Now we proceed to install it in a physical device (my own phone). The parameters we get are the real ones, and we'll take these for bypassing the check



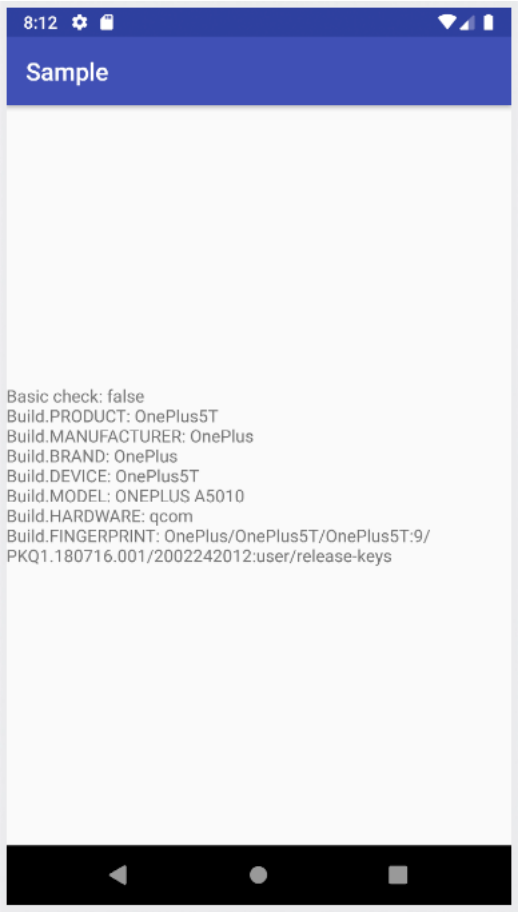
The following table is a summary of the parameters in both devices

Parameter	Emulator	Real Phone
Build.PRODUCT	sdk_gphone_x86_64	OnePlus5T
Build.MANUFACTURER	Google	OnePlus
Build.BRAND	google	OnePlus
Build.DEVICE	generic_x86_64	OnePlus5T
Build.MODEL	Android SDK built for x86_64	ONEPLUS A5010
Build.HARDWARE	ranchu	qcom
Build.FINGERPRINT	google/sdk_gphone_x86_64/generic_x86_64:9/PSR1.180720.093/5456446:userdebug/dev-keys	OnePlus/OnePlus5T/OnePlus5T:9/PKQ1.1807 keys

Same as before, we are using Frida to manipulate the check results. In this case the evasion is even better, since we are overriding directly the Android class `Build`. In this class methods are the properties described on the file. So we are not modifying the file, but the attributes of the class that parsed it. Because of this, this Frida script should be universal for all checks that use `andoroid.os.Build`, and it does not depend on the implementation of the app.

```
Java.perform(function () {
    var classname = "android.os.Build";
    var hookclass = Java.use(classname);

    hookclass.PRODUCT.value = "OnePlus5T";
    hookclass.PRODUCT.value = "OnePlus5T";
    hookclass.MANUFACTURER.value  = "OnePlus";
    hookclass.BRAND.value = "OnePlus";
    hookclass.DEVICE.value = "OnePlus5T";
    hookclass.MODEL.value = "ONEPLUS A5010";
    hookclass.HARDWARE.value = "qcom";
    hookclass.FINGERPRINT.value = "OnePlus/OnePlus5T/OnePlus5T:9/PKQ1.180716.001/2002242012:user/release-keys";

});
```

With this Frida script, we can see that the parameters read from the file are the fake ones, and that the check returns false.



Certificate pinning

Pinning is the process of associating a host with their expected certificate or public key. In an Android app environment, this is achieved at HTTP level. Most HTTP libraries support pinning a certificate, which works by hardcoding the certificate or the key into the code. From now on, the application only accepts connections from servers that provide the inserted certificate. In the end what we are doing is breaking the chain of trust of certificates, by only trusting the final certificate and not any root certificate issued by any CA or other institution. By doing this the developer increases the difficulty of intercepting HTTPS requests, since the only allowed connections will be to the owner of the pinned certificate, and this certificate cannot be easily changed by adding certificates in the OS, for example.

Man in the Middle attacks on Android apps

In Android there are 2 kinds of certificates, user and system. User certificates are used by the browsers to determine the security of websites visited. These user certificates can be created and modified with user permissions easily. When applications other than a browser check for the validity of a certificate in a connection,by default they only trust the system certificates. These come preinstalled in the device, and can only be modified in a rooted device. So the first step in intercepting requests from the application we are reversing is to insert our own certificate in the root certificate store.

Test environment

To test the succesfull bypass I have developed a test application (https://github.com/jserrats/certpin) with very simple functionality. It consists of two buttons, both make a HTTPS request to a public API (`https://swapi.dev/api/people/3/`) and displays the returning JSON. One button makes a usual HTTPS request (control)in order to check if the connection is made successfully. This way we can see if and to see if we are able to intercept and modify the request using Burp. The other one makes a pinned request, with the hash of the certificate of `swapi.dev`.

In the following snippets we can see the difference between a request made without and with pinning.respectively. The full code can be found on the github project.

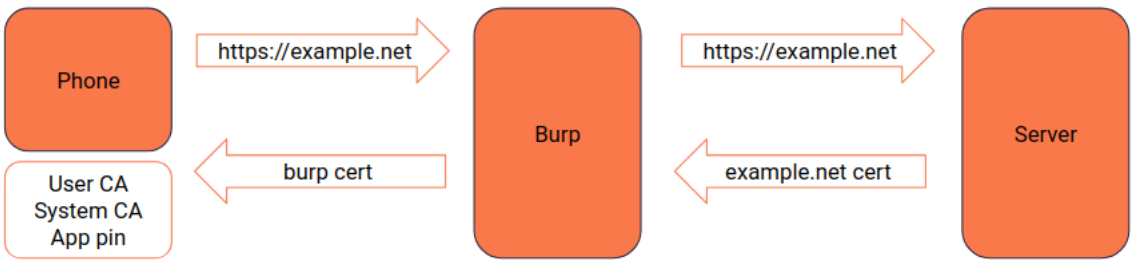
`PlainRequestActivity.java`

```
OkHttpClient client = new OkHttpClient();

        Request request = new Request.Builder()
                .url(url)
                .build();
```

`PinedRequestActity.java`

```
CertificatePinner certpin = new CertificatePinner.Builder()
                .add(hostname, "sha256/eSiyNwaNIbIkI94wfLFmhq8/ATxm30i973pMZ669tZo=")
                .build();

        OkHttpClient client = new OkHttpClient.Builder().certificatePinner(certpin)
                .build();

        Request request = new Request.Builder()
                .url(url)
                .build();
```

In the following figure we can see how the phone gets the Burp Proxy certificate, and why it is required that we insert its certificate in the System CA, before we can try evading the pinning.



This way we have four scenarios, with two tests to perform in each one:

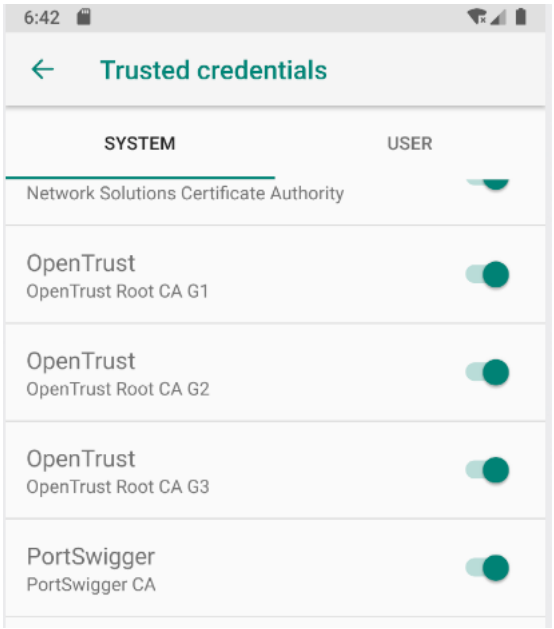
Scenario	Plain	Pinned
Normality	Successful	Successful
Burp - No certificate installed	Failure	Failure

Scenario	Plain	Pinned
Burp - Certificate installed	Successful	Failure
Burp - Certificate installed - Pin bypass enabled	Successful	Successful

Installing our root CA on the emulator

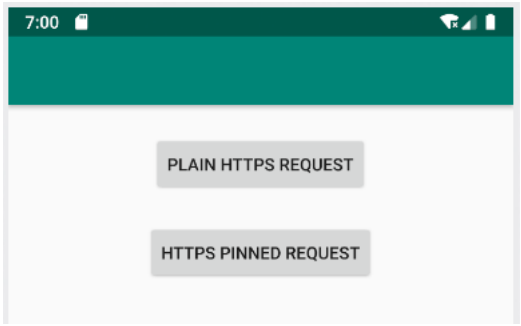
In the following steps i describe a way that I have particularly found to install a certificate on the system folder. Particularly we'll be installing the certificate from Burp Proxy

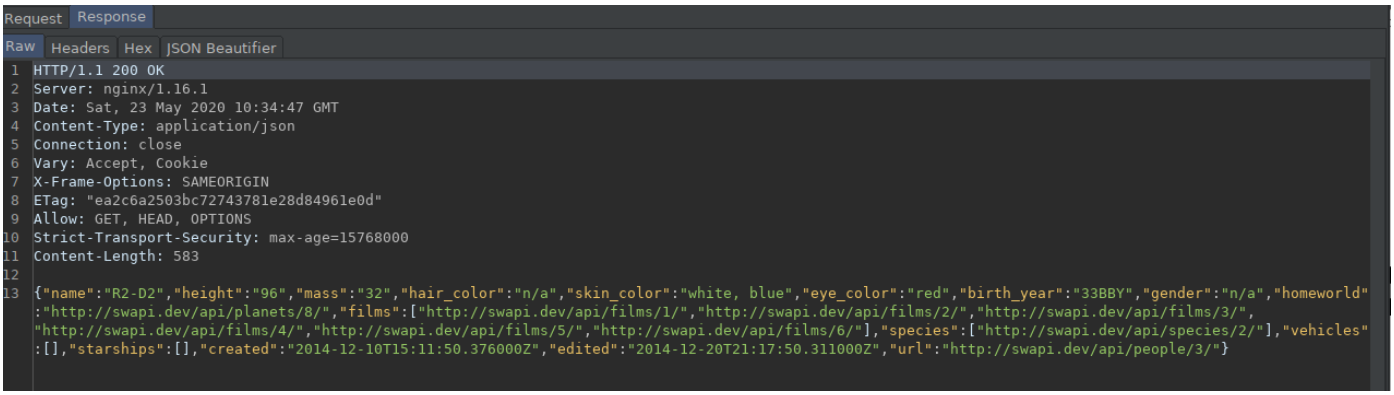
1. Obtain a writable `/system` partition
    1. Launch the emulator with the flag `-writable-system`
    2. Then obtain root on the system with `adb root`
    3. Then remount the `/system` partition as writeable with `adb remount`
    4. Test that the partition was mounted successfully with `touch /system/test.txt`
2. Obtain the Root CA from burp
    1. Obtain the Burp Root certificate. Enter `http://burp` into a browser configured with burp and download the CA file.
    2. Move the file to the SD card on the device with `adb push cacert.der /mnt/sdcard/cacert.cer`. In this step we are also changing the file's extension
3. Install the Root CA
    1. On the android emulator go to `Security & location > Advanced > Encryption & credentials > Install from SD card`
    2. Select the `cacert.cer` file from the SD. Put any name and choose the `VPN and Apps` option
4. Set the certificate as System certificate
    1. Copy the certificate to the correct folder in the system partition with `cp /data/misc/user/0/cacerts-added/* /system/etc/security/cacerts`. Now the device trusts the certificate and will be used in app connections

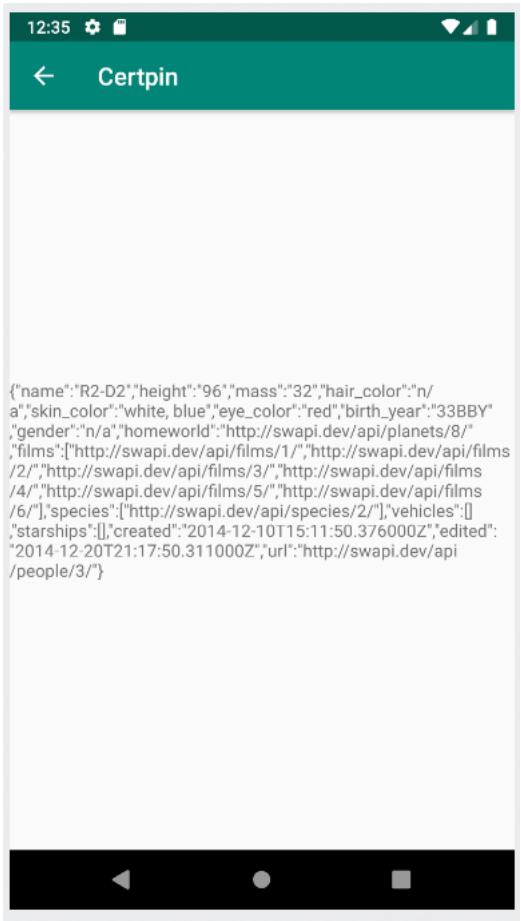


Using Burp Proxy to perform "Man in the Middle" attack

With the certificate installed we can configure the emulator to use Burp as a proxy in order to intercept requests. We can see the request appear in burp and in the app without any problem

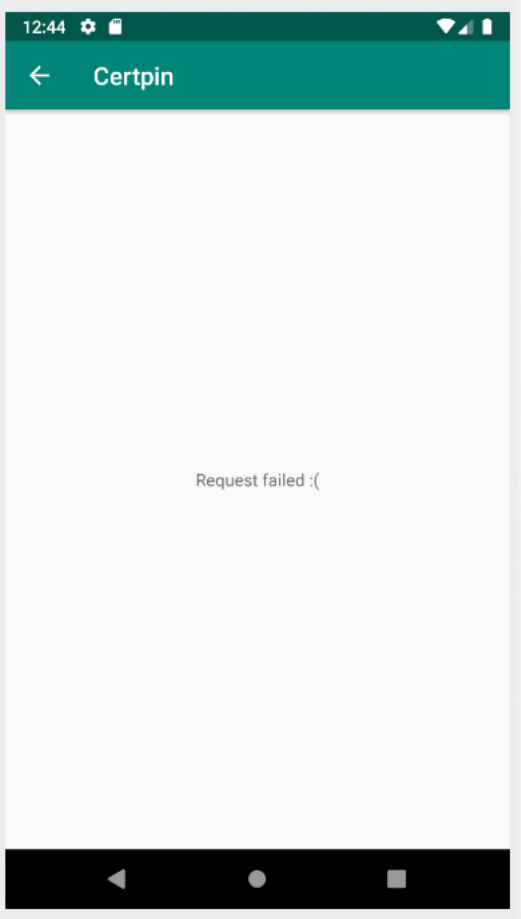






When performing a pinned request inside our app, the `Okhttp` library performs a check on the certificate provided. In this case, our installed certificate is not accepted by the app even if the system trusts it, as it does not pass the internal check

```
CertificatePinner certpin = new CertificatePinner.Builder()
         .add(hostname, "sha256/eSiyNwaNIbIkI94wfLFmhq8/ATxm30i973pMZ669tZo=")
         .build();
```



Bypassing certificate pinning

There are numerous ways to bypass a pinning, depending on how it was implemented. In this case there are two easy ways:

- Decompile the APK file, find the string with the hash of the CA, change it for the hash of our own app and repackage and sign the application
- Use Frida to disable the check performed by the HTTP library used (in this case OkHttp)

In this case we'll use the second method, as it is easier and more clean. To effectively disable the pinning check, we have to overwrite the method `check` from the class `CertificatePinner.java` from the OkHttp library. This information can be obtained from the own OkHttp documentation (https://square.github.io/okhttp/3.x/okhttp/okhttp3/CertificatePinner.html). Keep in mind that when this method finds a non valid certificate throws and exception, so to successfully bypass it we need to overwrite the whole execution of the method, not just override its return value.

```
setTimeout(function () {
    Java.perform(function () {
        // OkHTTPv3
        try {
            var okhttp3_Activity = Java.use('okhttp3.CertificatePinner');
            okhttp3_Activity.check.overload('java.lang.String', 'java.util.List').implementation = function (str) {
                console.log('[+] Bypassing OkHTTPv3 {1}: ' + str);
                return true;
            };

        } catch (err) {
            console.log('[-] OkHTTPv3 pinner not found');
            console.log(err);
        }

    });

}, 0);
```

Obfuscation

Obfuscation consists occurs during compilation, and consists on purposely complicating the code in a manner that the result functions exactly the same as the original, but is way harder to decompile and modify. This is achieved by complicating logic, encrypting resources, renaming methods and classes and adding superfluous code among others.

In an Android environment there are several tools that do obfuscation, but most of them (the supposedly better ones) require expensive licenses. Because of this in this project I'm using R8 which is free to use and has a high amount of documentation online.

R8

R8 is the default compiler that converts your project's Java bytecode into the DEX format that runs on the Android platform. This compiler is free and included by default in Android Studio.

While compiling, it can also be configured to do the following things:

- **Code shrinking (or tree-shaking):** detects and safely removes unused classes, fields, methods, and attributes from your app and its library dependencies.
- **Resource shrinking:** removes unused resources from your packaged app, including unused resources in your app's library dependencies.
- **Obfuscation:** shortens the name of classes and members, which results in reduced DEX file sizes.
- **Optimization:** inspects and rewrites your code to further reduce the size of your app's DEX files. For example, if R8 detects that the `else {}` branch for a given if/else statement is never taken, R8 removes the code for the `else {}` branch.

In this case, what will be affecting us is the obfuscation and optimization, which will remove the names of the functions and modify the flow of the code respectively.

Test environment

In order to understand how R8 applies obfuscation and how to bypass it, we are going to obfuscate the application developed in the previous chapter, certpin. Then we are going to see what has changed, how much more difficult it is now to bypass the pinning and how to do it now.

Build and install a release version

R8 functionalities are disabled in non release versions, because it can complicate debugging. In order to test the obfuscation we have to install a release version on the device. To do so, we simply have to set the option on `minifyEnabled` on the file `build.gradle`

```
buildTypes {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
```

and then choose a release build when launching gradle:

```
./gradlew assembleRelease
```

This will generate an `.apk` in the path `app/build/outputs/apk/release/app-release-unsigned.apk` file, but this cannot be installed directly on the device yet. Android requires `apk` files to be signed in order to install them. Luckily, any certificate is valid, so we can generate and autosign our own.

```
keytool -genkey -v -keystore my-release-key.keystore -alias alias_name -keyalg RSA -keysize 2048 -validity 10000
```

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore my-release-key.keystore app.apk alias_name
```

After this two simple steps, we'll have an application ready to be installed. By far the easiest way to do so is to use ADB.

```
adb install build.apk
```

After running this you should find the application installed in the android dashboard.

Decompile the apk

From now on we'll only work with the compiled APK. In order to obtain the maximum amount of information we'll both obtain the dex classes and use a java decompiler.

Apktool will give us a decompressed APK file with all the dalvik code obfuscated.

```
apktool d release/app-release-unsigned.apk -d releasedex
```

Jadx will try to decompile the dalvik code and give us an equivalent in to java. This is far from perfect, but it can be useful to understand some snippets of code.

```
jadx release/app-release-unsigned.apk
```

Here we can see a comparison between the same class `PinnedRequest.java` in the source code and the decompiled version after going trough R8 obfuscation

```java
public class PinnedRequestActivity extends AppCompatActivity {

    TextView txtString;
    public String hostname = "swapi.dev";
    public String url = "https://swapi.dev/api/people/3/";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_plain_request);
        txtString = (TextView) findViewById(R.id.txtString);

        try {
            run();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    void run() throws IOException {

        //OkHttpClient client = new OkHttpClient();

        CertificatePinner certpin = new CertificatePinner.Builder()
                .add(hostname, "sha256/eSiyNwaNIbIkI94wfLFmhq8/ATxm30i973pMZ669tZo=")
                .build();

        OkHttpClient client = new OkHttpClient.Builder().certificatePinner(certpin)
                .build();

        Request request = new Request.Builder()
                .url(url)
                .build();

        client.newCall(request).enqueue(new Callback() {
            @Override
            public void onFailure(Call call, IOException e) {
                Log.d("DEBUG", "Request Failed");
                call.cancel();
                PinnedRequestActivity.this.runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        txtString.setText("Request failed :(");
                    }
                });
                Log.e("ERROR", e.toString());
            }

            @Override
            public void onResponse(Call call, Response response) throws IOException {

                final String myResponse = response.body().string();
                Log.d("DEBUG", myResponse);
                PinnedRequestActivity.this.runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        txtString.setText(myResponse);
                    }
                });

            }
        });
```

```
    }
}
```

```
public class PinnedRequestActivity extends j {
    public TextView r;
    public String s = "swapi.dev";
    public String t = "https://swapi.dev/api/people/3/";

    public void n() {
        Set set;
        int i;
        ArrayList arrayList = new ArrayList();
        String str = this.s;
        String[] strArr = {"sha256/eSiyNwaNIbIkI94wfLFmhq8/ATxm30i973pMZ669tZo="};
        if (str != null) {
            for (String str2 : strArr) {
                arrayList.add(C0160g.f2573b.a(str, str2));
            }
            int size = arrayList.size();
            if (size == 0) {
                set = k.f2078a;
            } else if (size != 1) {
                int size2 = arrayList.size();
                if (size2 < 3) {
                    i = size2 + 1;
                } else {
                    i = size2 < 1073741824 ? size2 + (size2 / 3) : Integer.MAX_VALUE;
                }
                set = new LinkedHashSet(i);
                g.a(arrayList, set);
            } else {
                set = Collections.singleton(arrayList.get(0));
                e.a((Object) set, "java.util.Collections.singleton(element)");
            }
            C0160g gVar = new C0160g(set, null);
            z.a aVar = new z.a();
            aVar.u = gVar;
            z zVar = new z(aVar);
            C.a aVar2 = new C.a();
            aVar2.b(this.t);
            B.a(zVar, aVar2.a(), false).a(new c(this));
            return;
        }
        e.a("pattern");
        throw null;
    }

    @Override // a.h.a.f, a.a.c, a.b.a.j, a.k.a.ActivityC0086i
    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.activity_plain_request);
        this.r = (TextView) findViewById(R.id.txtString);
        try {
            n();
        } catch (IOException e2) {
            e2.printStackTrace();
        }
    }
}
```

It is quite clear now that this process has complicated enormously the task of understanding what this code is doing, as any names given are replaced by random characters, and the general flow is much more difficult. Notice that the strings are intact. Paid obfuscators like Dexguard do encrypt strings among other assets in order to make it even more difficult to understand what the code does. Since R8 main objective is not security, the obfuscation it does is clearly not enough, though it will be a challenge anyway.

Find new name for pinner class

By looking at the decompiled `PinnedRequestActivity.java` class (whose name was not modified because it appears in the Android manifest) we can reach the conclusion that the method `run()` is now renamed to `n()`. The string containing the sha256 of the certificate is a great indicator. A quick solution would be to simply modify this string and replace it with the hash of our certificate, and then repack and resign the application. This approach would not work if the release version was obfuscated with a professional grade tool, such as dexguard, since it does encrypt all plain strings found in code. Because of this, we'll try to find the new name of the class and bypass the pinning using Frida once again.

```
public void n() {
        Set set;
        int i;
        ArrayList arrayList = new ArrayList();
        String str = this.s;
        String[] strArr = {"sha256/eSiyNwaNIbIkI94wfLFmhq8/ATxm30i973pMZ669tZo="};
        if (str != null) {
            for (String str2 : strArr) {
                arrayList.add(C0160g.f2573b.a(str, str2));
            }
            int size = arrayList.size();
            if (size == 0) {
                set = k.f2078a;
            } else if (size != 1) {
                int size2 = arrayList.size();
                if (size2 < 3) {
                    i = size2 + 1;
                } else {
                    i = size2 < 1073741824 ? size2 + (size2 / 3) : Integer.MAX_VALUE;
                }
                set = new LinkedHashSet(i);
                g.a(arrayList, set);
            } else {
                set = Collections.singleton(arrayList.get(0));
                e.a((Object) set, "java.util.Collections.singleton(element)");
            }
            C0160g gVar = new C0160g(set, null);
            z.a aVar = new z.a();
            aVar.u = gVar;
            z zVar = new z(aVar);
            C.a aVar2 = new C.a();
            aVar2.b(this.t);
            B.a(zVar, aVar2.a(), false).a(new c(this));
            return;
        }
        e.a("pattern");
        throw null;
    }
```

Then we can go to the Dalvik code obtained with Apktool, search for the method `n()` and follow the code flow in order to see where the hash of the certificate ends up.

```
# virtual methods
.method public n()V
    .locals 9

    .line 1
    new-instance v0, Ljava/util/ArrayList;

    invoke-direct {v0}, Ljava/util/ArrayList;-><init>()V

    .line 2
    iget-object v1, p0, Lxyz/jserrats/certpin/PinnedRequestActivity;->s:Ljava/lang/String;

    const-string v2, "sha256/eSiyNwaNIbIkI94wfLFmhq8/ATxm30i973pMZ669tZo="

    filled-new-array {v2}, [Ljava/lang/String;

    move-result-object v2

    const/4 v3, 0x0

    if-eqz v1, :cond_5

    .line 3
    array-length v4, v2

    const/4 v5, 0x0

    move v6, v5

    :goto_0
    if-ge v6, v4, :cond_0

    aget-object v7, v2, v6
```

```
    sget-object v8, Ld/g;->b:Ld/g$b;

    invoke-virtual {v8, v1, v7}, Ld/g$b;->a(Ljava/lang/String;Ljava/lang/String;)Ld/g$c;

    move-result-object v7

    invoke-interface {v0, v7}, Ljava/util/List;->add(Ljava/lang/Object;)Z

    add-int/lit8 v6, v6, 0x1

    goto :goto_0

    //...
```

In this line we can see that we are invoking the method `a` from the class `d.g`. This line is equivalent to this other one in the source code:

```
invoke-virtual {v8, v1, v7}, Ld/g$b;->a(Ljava/lang/String;Ljava/lang/String;)Ld/g$c;
```

```
        CertificatePinner certpin = new CertificatePinner.Builder()
                .add(hostname, "sha256/eSiyNwaNIbIkI94wfLFmhq8/ATxm30i973pMZ669tZo=")
                .build();
```

This method is invoked passing the hash of the certificate, and has the same signature `"java.lang.String","java.lang.String"`, so we can deduce the class CertificatePinner has the name `d.g`.

If we enumerate classes which package begin with `d.g` and then analyze their signatures to one that matches the method `check(String hostname, List<Certificate> peerCertificates)` we'll find the class we are looking for.

If we look for the implementation of this CertificatePinner

> Confirms that at least one of the certificates pinned for hostname is in peerCertificates. Does nothing if there are no certificates pinned for hostname. OkHttp calls this after a successful TLS handshake, but before the connection is used.
>
> Throws:
> `SSLPeerUnverifiedException` - if peerCertificates don't match the certificates pinned for hostname.

So in order to bypass this control, we simply have to not execute this function in order to avoid raising any exception.

The Frida script is exactly the same as the one used in the certificate pinning excercise, but with the modified path and function name.

```
Java.perform(function () {
    var classname = "d.g";
    var classmethod = "a";
    var hookclass = Java.use(classname);

    hookclass.a.overload("java.lang.String","java.util.List").implementation = function (v0,v1) {
        send("CALLED: " + classname + "." + classmethod + "()\n");

        var s="";
        s=s+"HOOK: " + classname + "." + classmethod + "()\n";
        s=s+"IN: "+eval(v0,v1)+"\n";
        s=s+"OUT: "+ret+"\n";

        send(s);

        return ret;
    };
});
```

Results

Through the course of this project I have learnt a great amount about the basics of application security development on Android. While the level of insight achieved on this project it is not enough to be applied to critical applications that deal with sensible information, I think it sets a foundation on how this subject works, and it can be easily expanded from here.

As a result I have also published this document on github hoping it can be useful to any developer or analyst assessing the security of an application.

Conclusions

Reverse engineer protections are a race against the attacker's time and resources. Ultimately, the attacker always given infinite resources. The objective here is to make it difficult enough so it is not worth it for the attacker spend time reversing this application. Also new technologies, controls and evasions appear every day, so it is also important to keep the protections updated and follow the news on the latest technologies.

Future work could try to include other protections such as integrity check, hooking detection or others, or try to evade more advanced protections.

Bibliography

- OWASP Mobile Security Testing Guide (MSTG) - https://github.com/OWASP/owasp-mstg
- OWASP Mobile Application Security Verification Standard (MASVS) - https://github.com/OWASP/owasp-masvs
- Decompiling Android - Godfrey Nolan
- Android Security Internals - Nikolay Elenkov